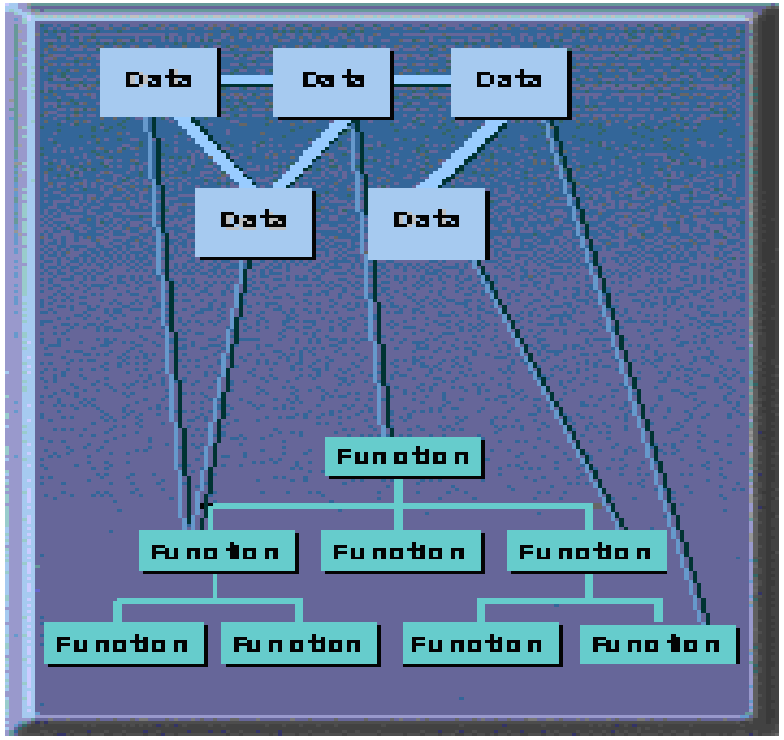
TATA CONSULTANCY SERVICES

TATA

Object Oriented Programming in ABAP

An Overview

Procedural Programming



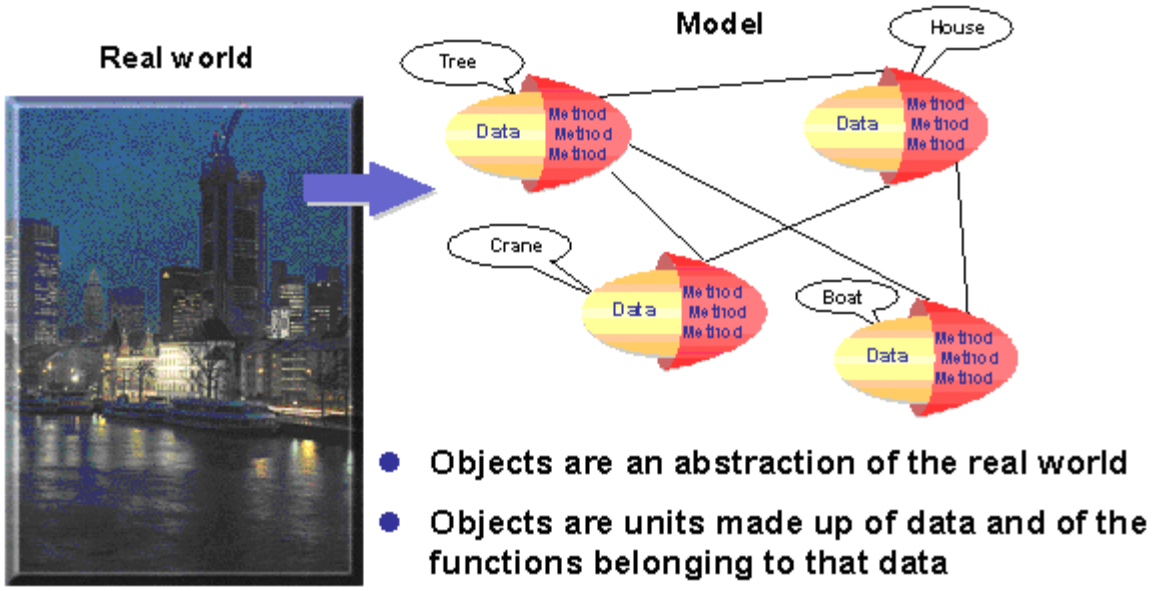
- Information systems used to be defined primarily by their functions: data and functions were stored separately and linked using input-output relationships.
- Direct access to data

Structure of an ABAP Program

```
TYPES: ...

DATA: ...

...

PERFORM f1 ...

CALL FUNCTION ...

...



FORM f1 ...
   ...
ENDFORM.
```

- **Data declaration**
- **Main program**
  - **Call subroutines**
  - **Call function modules**
- **Define subroutines**

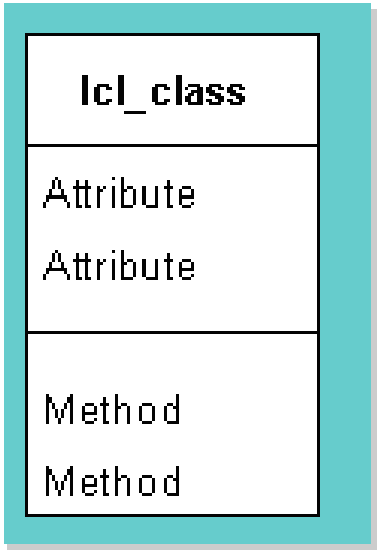
Objects



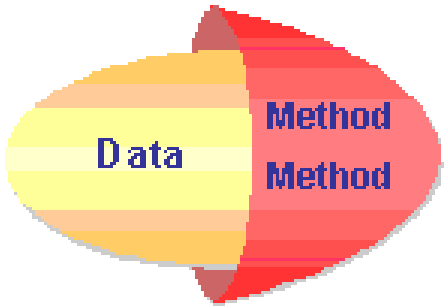
- Object orientation focuses on objects that represent either abstract or concrete things in the real world. They are first viewed in terms of their characteristics, which are mapped using the object's internal structure and attributes (data). The behaviour of an object is described through methods and events (functionality).
- Objects form capsules containing the data itself and the behaviour of that data. Objects should enable you to draft a software solution that is a one-to-one mapping of the real-life problem area.

Object-oriented Programming Model

- Class
  - Gives a general description of objects ("blueprint")
  - Establishes status types (attributes) and behavior (methods)



- Object
  - Reflection of real world
  - Specific instance of a class



Advantages of Object-oriented Approach

- **Consistency throughout the software development process**

  The “language” used in the various phases of software development (analysis, specification, design and implementation) is uniform. The ideal would be for changes made during the implementation phase to flow back into the design automatically.

- **Encapsulation**

  Encapsulation means that the implementation of an object is hidden from other components in the system, so that they cannot make assumptions about the internal status of the object and therefore dependencies on specific implementations do not arise.

Advantages of Object-oriented Approach

- **Polymorphism**

  Polymorphism (ability to have multiple forms) in the context of object technology signifies that objects in different classes have different reactions to the same message.

- **Inheritance**

  Inheritance defines the implementation relationship between classes, in which one class (the subclass) shares the structure and the behaviour defined in one or more other classes (superclasses).

  Note: ABAP objects only allows single inheritance.

ABAP Objects

- True, compatible extension of ABAP
- ABAP Objects statements can be used in "conventional" ABAP programs
- ABAP statements can be used in ABAP Objects programs

```
* ABAP Program

CLASS lcl_airplane DEFINITION.
  ...
ENDCLASS.
...
TYPES: ...
DATA: ...
...
```

```
* ABAP Objects Program

DATA: counter TYPE i.
...
CREATE OBJECT ...
counter = counter + 1.
...
```

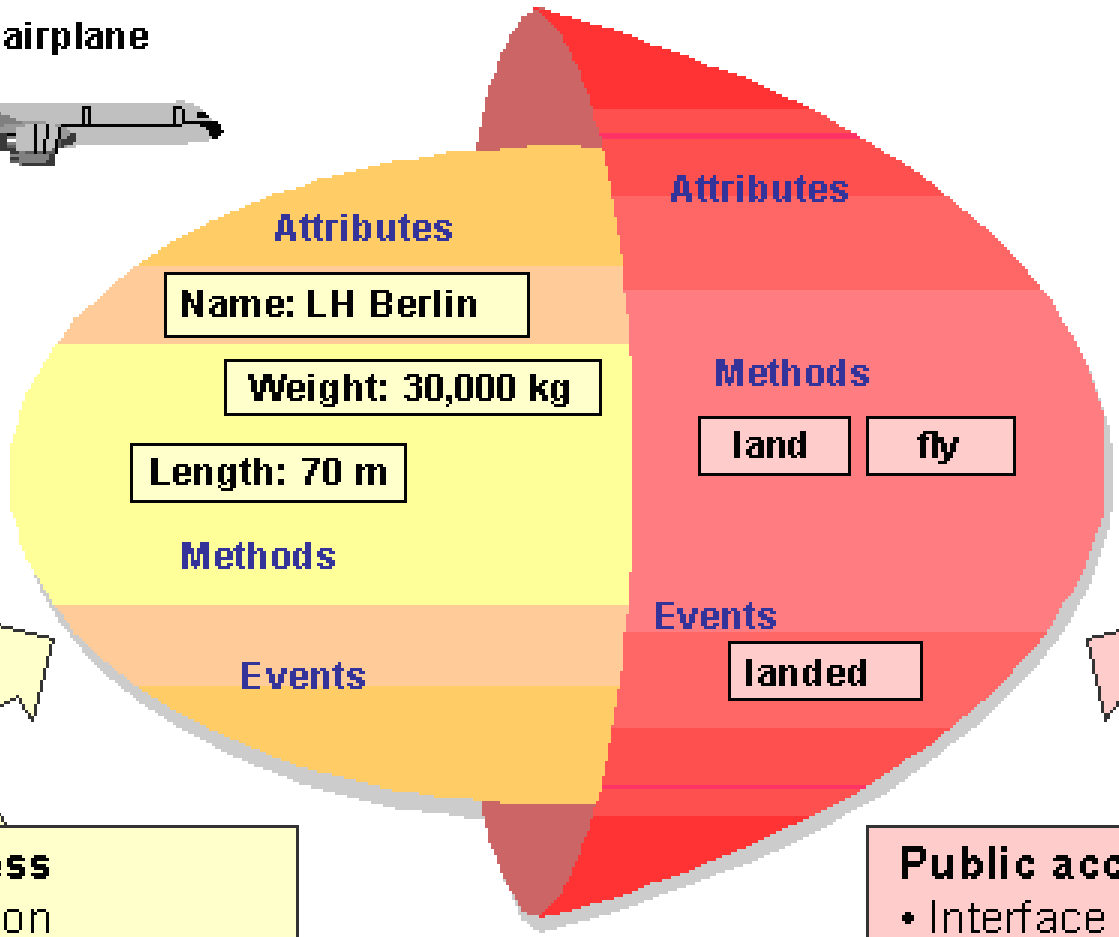
- ABAP objects is not a new language, but has been developed as an extension of ABAP. It integrates seamlessly into ABAP syntax and the ABAP programming model. All enhancements are strictly upward compatible.

Principles

- Objects
- Classes
- Attributes
- Methods
- Instantiation, garbage collector
- Working with objects
- Further principles

Objects

Example: airplane



Private access
- Encapsulation
- As a rule, attributes

Public access
- Interface
- As a rule,methods, events

Objects

- The object in the model has two layers: an outer shell and an inner core. Users can only see the outer shell, while the inner core remains hidden (the internal status of an object can only be seen within the object itself).

- Public components (outer shell): the outer shell contains the components of the object that are visible to users, such as attributes (data), methods (functions) and events. All users have direct access to these components. The public components of an object form its external point of contact.

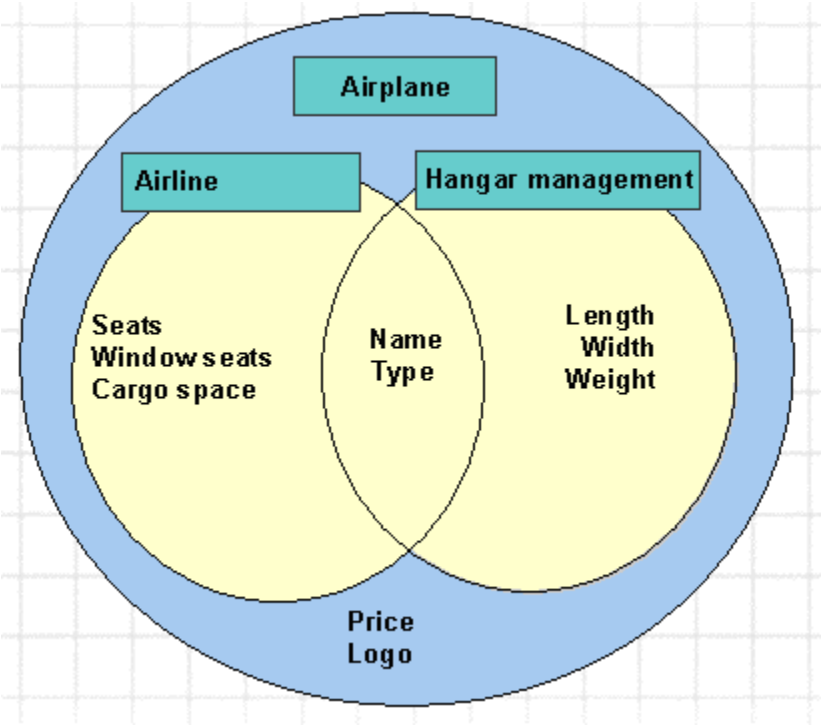
- Private components (inner core): the components of the inner core (attributes, methods and events) are only visible within the object itself. The attributes of an object are generally private. These private attributes of an object can only be accessed using the methods of that object itself.

Objects

Why are the private components of an object “hidden”?

- This principle is called “information hiding” or “encapsulation” and is used to protect the user.
- Let us assume that an object changes its private components, while its external point of contact remains unchanged. Any user who simply needs to access the object’s external point of contact can carry on working with the object as usual. The user does not notice the change.
- However, if an object changes its public components, then any user who accesses these public components must take these changes into account.

Classes



- In this context, abstractions are a simplified representations of complex relationships in the real world. An actually existing object is abstracted to the significant dimensions that are to be mapped. Insignificant details are left out in order to aid understanding of the overall system.
- This example concerns airplanes. Software for airlines and software for an airport's hangar management contain different abstractions (classes) for these objects.
- Important components – attributes, methods

Important Components in a Class

- Attributes
  - Data
  - Determine the state of the Object
- Methods
  - Executable Coding
  - Determine the behavior of the Object

Attributes

- Attributes describe the data that can be stored in the objects in a class.
- Class attributes can be of any type:
  - Data types: scalar (for example, data element), structured, in tables
  - ABAP elementary types (C, I, ...)
  - Object references
  - Interface references
- Attributes of the airplane class are, for example:
  - Name
  - Seats
  - Weight
  - Length
  - Wings
  - Tank

Public and Private Attributes

- You can protect attributes against access from outside by characterizing them as private attributes (defined in the PRIVATE SECTION).
- Attributes and their values that may be used directly by an external user are public attributes and are defined in the PUBLIC SECTION.
- Public attributes belong to the class ‘external point of contact’ that is, their implementation is publicized. If you want to hide the internal implementation from users, you must define internal and external views of attributes.
- As a general rule, you should define as few public attributes as possible.

Static Attributes and Instance Attributes

- There are two kinds of attributes
  - Static attributes
  - Instance attributes
- Instance attributes are attributes that exist separately for each object.
- Instance attributes are defined using the DATA keyword.
- Static attributes exist once only for each class and are visible for all (runtime) instances in that class. Static attributes usually contain information that is common to all instances, such as:
  - Data that is the same in all instances
  - Administrative information about the instances in that class (for example, counters and so on)
  - Static attributes are defined using the CLASS-DATA keyword.
- You may come across the expression “class attributes” in documentation, however, the official term in ABAP objects (as in C++, java) is “static” attributes.

Methods

- Methods are internal procedures in classes that determine the behaviour of an object. They can access all attributes in their class and can therefore change the state of an object.
- Methods have a parameter interface that enables them to receive values when they are called and pass values back to the calling program.
- In ABAP objects, methods can have IMPORTING, EXPORTING, CHANGING and RETURNING parameters as well as EXCEPTIONS. All parameters can be passed by value or reference.

Methods

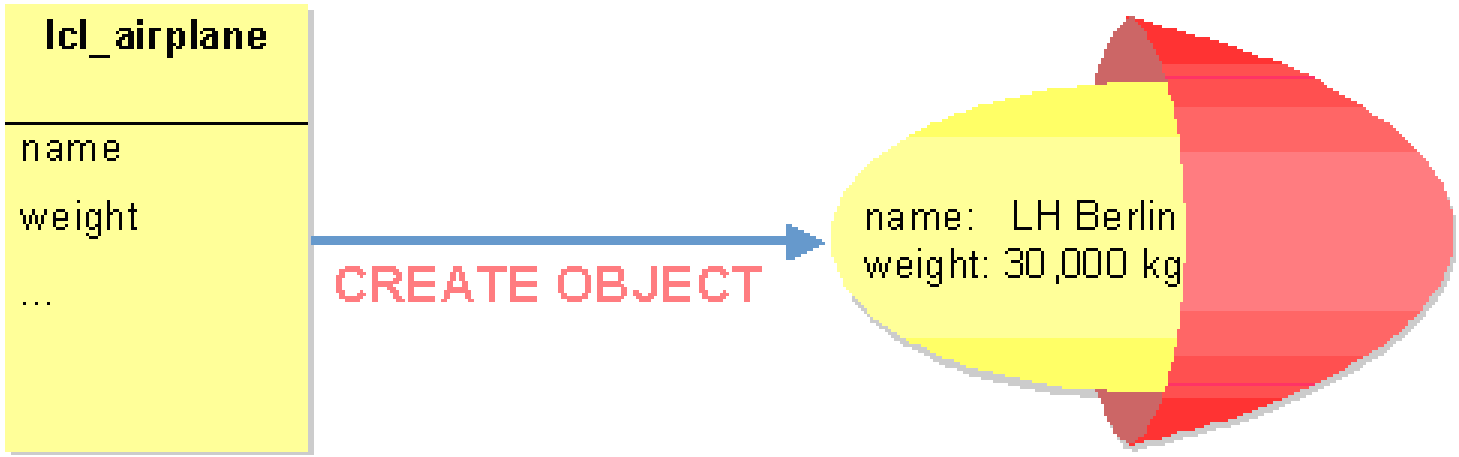
- You can define a return code for methods using RETURNING. You can only do this for a single parameter, which additionally must be passed as a value. Also, you cannot then define EXPORTING and CHANGING parameters. You can define functional methods using the RETURNING parameter (explained in more detail below).
- All input parameters (IMPORTING, CHANGING parameters) can be defined as optional parameters in the declaration using the OPTIONAL or DEFAULT additions. These parameters then do not necessarily have to be passed when the object is called. If you use the OPTIONAL addition, the parameter remains initialized according to type, whereas the DEFAULT addition allows you to enter a start value.

Static Methods and Instance Methods

- Static methods are defined on the class level. They are similar to instance methods, but with the restriction that they can only use static components (such as static attributes) in the implementation part. This means that static methods do not need instances and can therefore be called from anywhere. They are defined using the CLASS-METHODS statement, and they are bound by the same syntax and parameter rules as instance methods.
- The term “class method” is common, but the official term in ABAP objects (as in C++, java) is “static method”.

Creating Objects

- **Objects can only be created and addressed using reference variables**



Creating Objects

- A class contains the generic description of an object. It describes all the characteristics that are common to all the objects in that class. During the program runtime, the class is used to create specific objects (instances). This process is called instantiation.
- Example:

  The object LH Berlin is created during runtime in the main memory by instantiation from the lcl_airplane class.

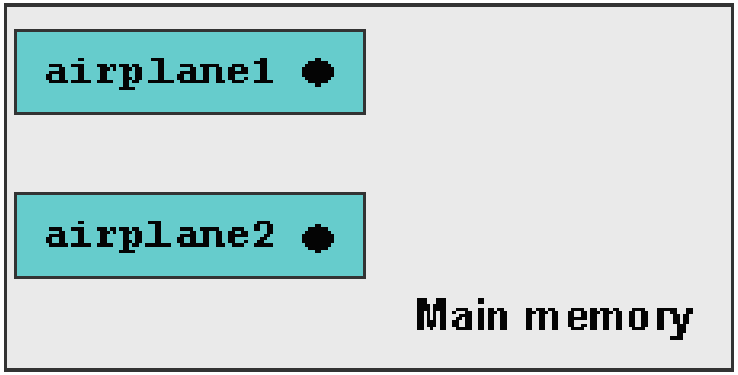
  The lcl_airplane class itself does not exist as an independent runtime object in ABAP objects.
- Realization:

  Objects are instantiated using the statement: CREATE OBJECT.

  During instantiation, the runtime environment dynamically requests main memory space and assigns it to the object.

Reference Variables

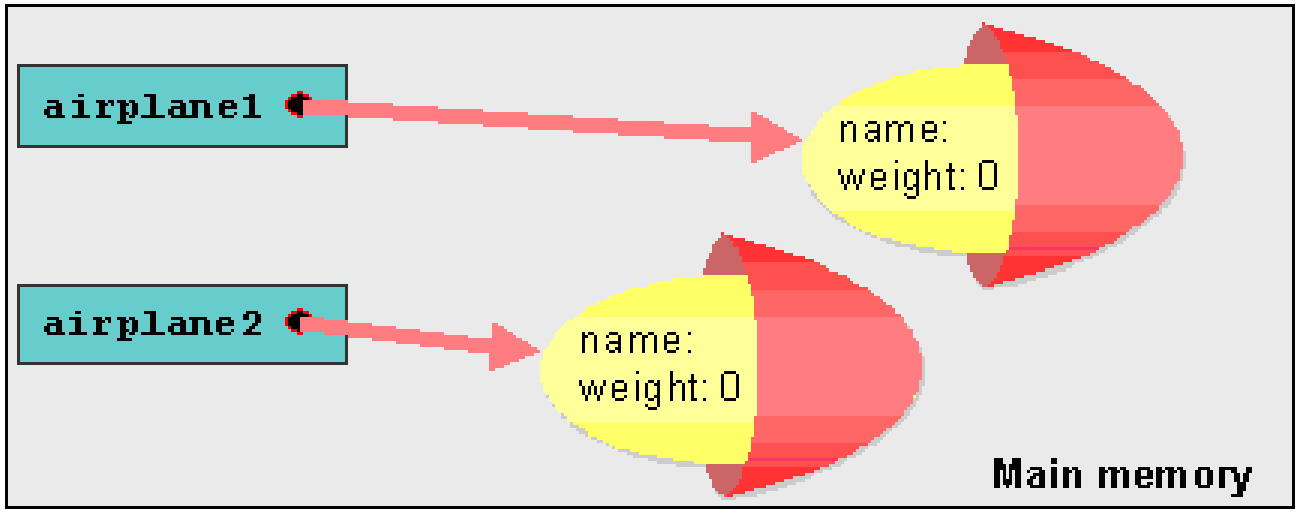
```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    ...
  PRIVATE SECTION.
    ...
ENDCLASS.


CLASS lcl_airplane IMPLEMENTATION.
  ...
ENDCLASS.


DATA: airplane1 TYPE REF TO cl_airplane,
      airplane2 TYPE REF TO cl_airplane.
```



Creating Objects: Syntax

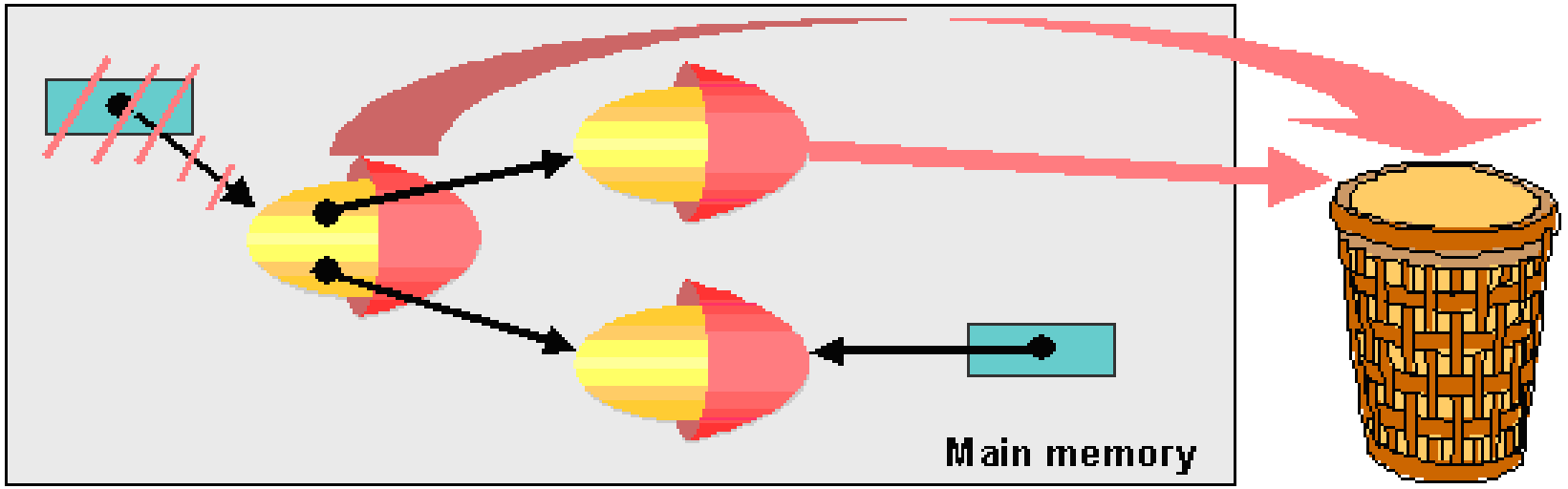
```
CREATE OBJECT <reference>.
```

```
DATA: airplane1 TYPE REF TO lcl_airplane,
      airplane2 TYPE REF TO lcl_airplane.

CREATE OBJECT airplane1.
CREATE OBJECT airplane2.
```



Garbage Collector Concept

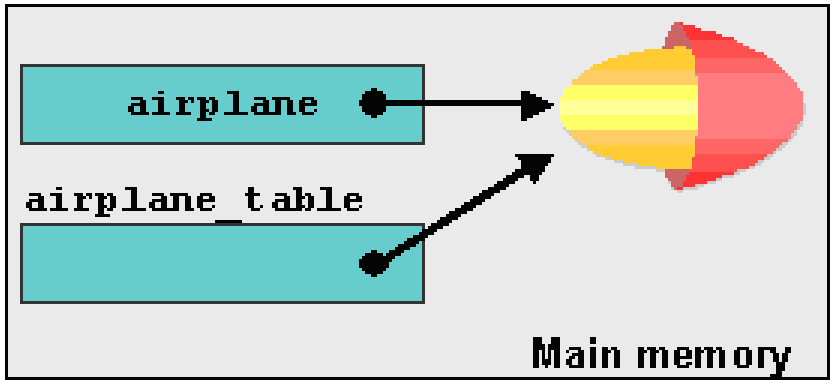
- All independent references in the global main memory are checked. The references point to active objects, which are marked internally.
- If class or instance attribute references point to other objects, these are also marked.
- Objects that are not marked are deleted from the main memory.



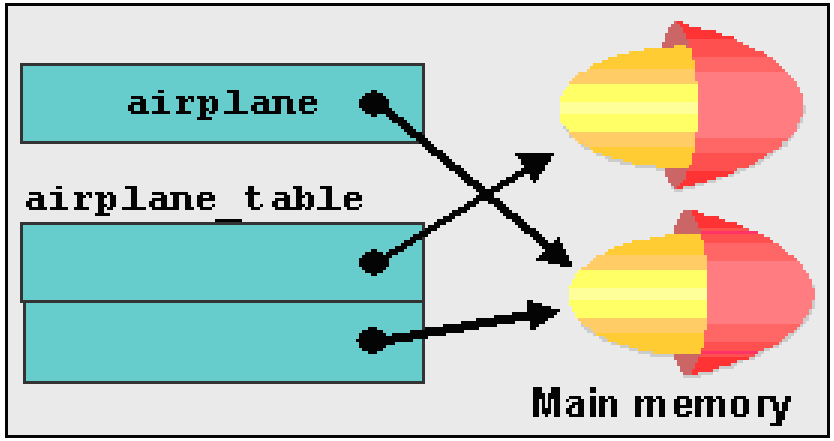
Object Identity

```
DATA: airplane          TYPE REF TO cl_airplane,
      airplane_table    TYPE TABLE OF REF TO cl_airplane.
```

```
CREATE OBJECT airplane.
APPEND airplane TO airplane_table.
```



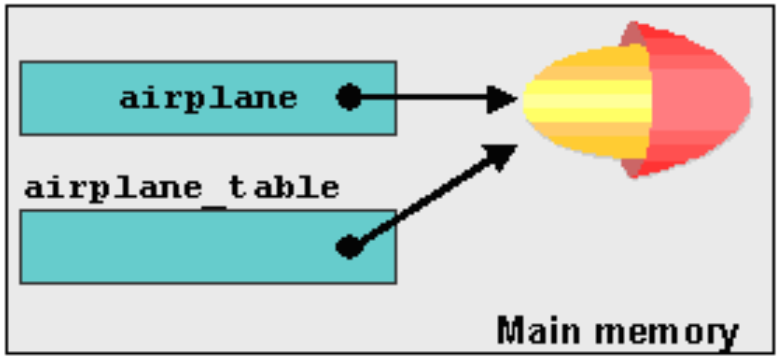

```
CREATE OBJECT airplane.
APPEND airplane TO airplane_table.
```



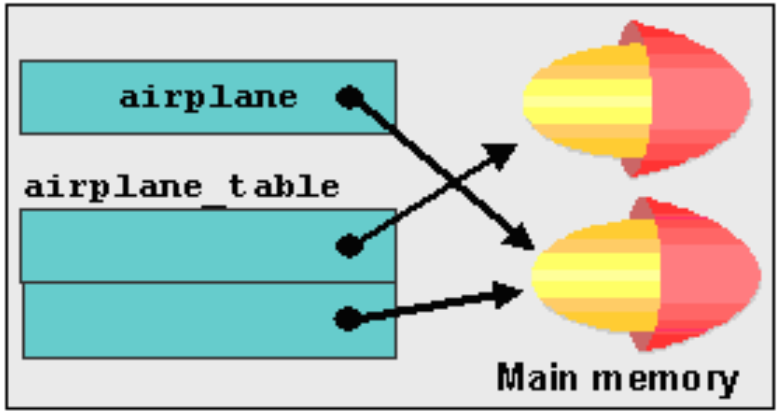
Assigning References

```
DATA: airplane          TYPE REF TO cl_airplane,
      airplane_table    TYPE TABLE OF REF TO cl_airplane.
```

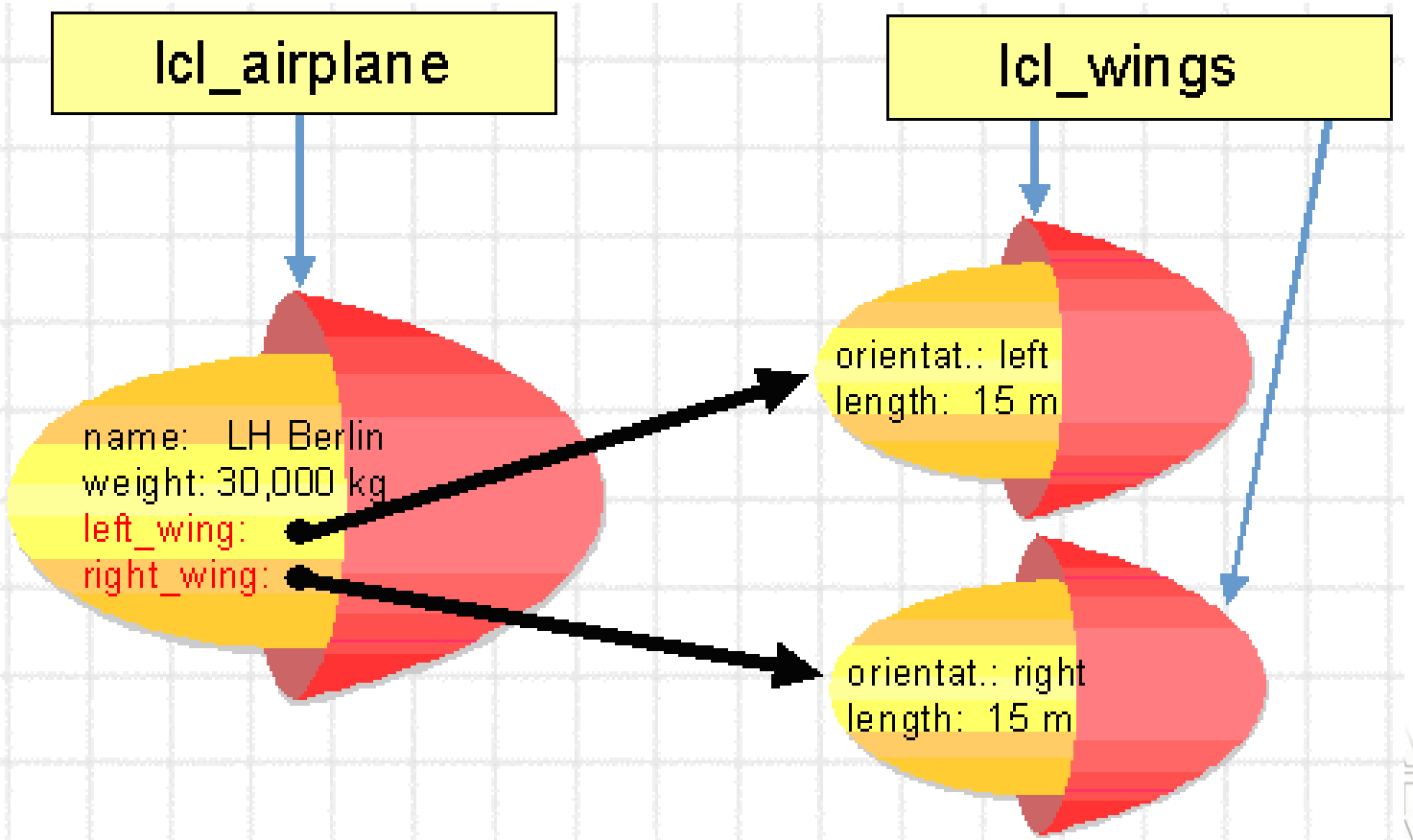
```
CREATE OBJECT airplane.
APPEND airplane TO airplane_table.
```




```
CREATE OBJECT airplane.
APPEND airplane TO airplane_table.
```



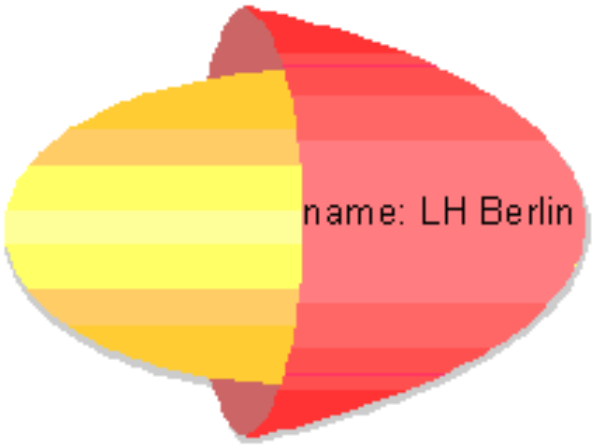
Object References As Attributes



External Access to Public Attributes

Instance attribute:
<reference>-><instance_attribute>

Class attribute:
<classname>=><class_attribute>




```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    DATA: name    TYPE string READ-ONLY.
    CLASS-DATA: count TYPE I READ-ONLY.
    ...
ENDCLASS.
...

DATA: airplane1 TYPE REF TO lcl_airplane.
DATA: airplane_name TYPE STRING,
      n_o_airplanes TYPE i.
...

airplane_name = airplane1->name.
n_o_airplanes = lcl_airplane=>count.
```

Calling Methods

- Every object behaves in a certain way. This behavior is determined by its methods. There are three types of method:
  - 1. Methods that cause behavior and do not pass values
  - 2. Methods that pass a value
  - 3. Methods that pass or change several values
- An object that requires services from another object sends a message to the object providing the services. This message names the operation to be executed. The implementation of this operation is known as a method.
- Public methods can be called from outside the class in a number of ways: instance methods are called using CALL METHOD <reference> -><instance_method>.

  Static methods are called using CALL METHOD classname>=><class_method>.

  Static methods are addressed by class name, since they do not need instances.

Calling Methods

Instance methods:

```
CALL METHOD <instance>-><instance_method>
            EXPORTING <im_var> = <variable>
            IMPORTING <ex_var> = <variable>
            CHANGING  <ch_var> = <variable>
            RECEIVING <re_var> = <variable>
            EXCEPTIONS <exception> = <nr>.
```

Static methods:

```
CALL METHOD <classname>=><class_method>
            EXPORTING ...  .
```

```
DATA: airplane TYPE REF TO lcl_airplane.
DATA: name TYPE string.
DATA: count_planes TYPE I.

CREATE OBJECT airplane.

CALL METHOD airplane->set_name EXPORTING im_name = name.
CALL METHOD lcl_airplane=>get_count RECEIVING re_count = count_planes.
```

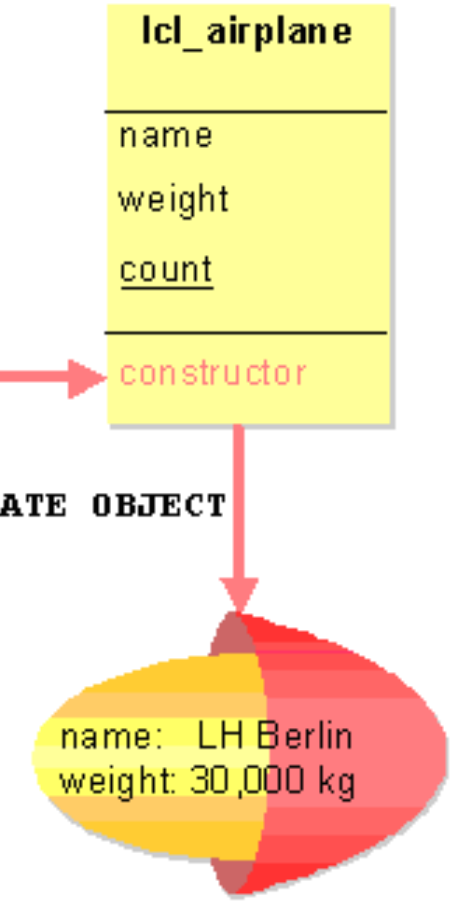
Functional Methods

- Methods that have a RETURNING parameter are described as functional methods. These methods cannot have EXPORTING or CHANGING parameters, but has many (or as few) IMPORTING parameters and EXCEPTIONS as required.
- Functional methods can be used directly in various expressions (although EXCEPTIONS are not catchable at the moment - you must use the long form of the method call):
  - In logical expressions (IF, ELSEIF, WHILE, CHECK, WAIT)
  - In the CASE statement (CASE, WHEN)
  - In the LOOP statement
  - In arithmetic expressions (COMPUTE)
  - In bit expressions (COMPUTE)
  - In the MOVE statement.

Constructor

- Special method for creating objects with defined initial state
- Only has IMPORTING parameters and EXCEPTIONS
- Exactly one constructor is defined per class (explicitly or implicitly)
- Is executed exactly once per instance

```
METHODS CONSTRUCTOR IMPORTING <im_parameter>
                    EXCEPTIONS <exception>.
```



Constructor

- The constructor is a special (instance) method in a class and is always named CONSTRUCTOR. The following rules apply:
  - Each class has exactly one constructor.
  - The constructor does not need to be defined if no implementation is defined.
  - The constructor is automatically called during runtime within the CREATE OBJECT statement.
  - If you need to implement the constructor, then you must define and implement it in the PUBLIC SECTION.
- When EXCEPTIONS are triggered in the constructor, instances are not created (as of 4.6a), so no main memory space is taken up.

Constructor: Example

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS CONSTRUCTOR IMPORTING im_name   TYPE string
                                  im_weight TYPE I.
  PRIVATE SECTION.
    DATA: name  TYPE string, weight TYPE I.
    CLASS-DATA count TYPE I.
ENDCLASS.
```

```
CLASS lcl_airplane IMPLEMENTATION.
  METHOD CONSTRUCTOR.
    name   = im_name.
    weight = im_weight.
    count = count + 1.
  ENDMETHOD.
ENDCLASS.
```

```
DATA airplane TYPE REF TO lcl_airplane.
...
CREATE OBJECT airplane
              EXPORTING im_name   = `LH Berlin
                        im_weight = 30000.
```

Static Constructor

- The static constructor is a special static method in a class and is always named CLASS_CONSTRUCTOR. It is executed precisely once per program. The static constructor of class <classname> is called automatically before the class is first accessed, that is, before any of the following actions are executed:
  - Creating an instance in the class using CREATE OBJECT obj, where obj has the data type

    REF TO <classname>.
  - Addressing a static attribute using <classname>=><an_attribute>.
  - Calling a static attribute using CALL METHOD <classname>=><a_classmethod>.
  - Registering a static event handler method using SET HANDLER <classname>=><handler_method> for obj.
  - Registering an event handler method for a static event in class <classname>.
- The static constructor cannot be called explicitly.

Encapsulation

- The principle of encapsulation is to hide the implementation of a class from other components in the system, so that these components cannot make assumptions about the internal state of the objects in that class or of the class itself. This prevents dependencies on specific implementations from arising.
- The class is the capsule surrounding related functions.
- The principle of visibility ensures that the implementation of the functions and the information administered within a class is hidden.

Delegation Principal

- In delegation, two objects are involved in handling a request: the recipient of the request delegates the execution of the request to a delegate.
- The main advantage of delegation (as a re-use mechanism) lies in the option of changing the behavior of the recipient by substituting the delegate (at runtime). For example, delegation enables the airplane to be equipped with a new tank, without the call changing for the client or for the airplane class.
- Good capsulation often forces you to use delegation: if tank in the above example were a private attribute in class lcl_airplane, then the user cannot address the tank directly, but only through the airplane!

Namespace Within Class

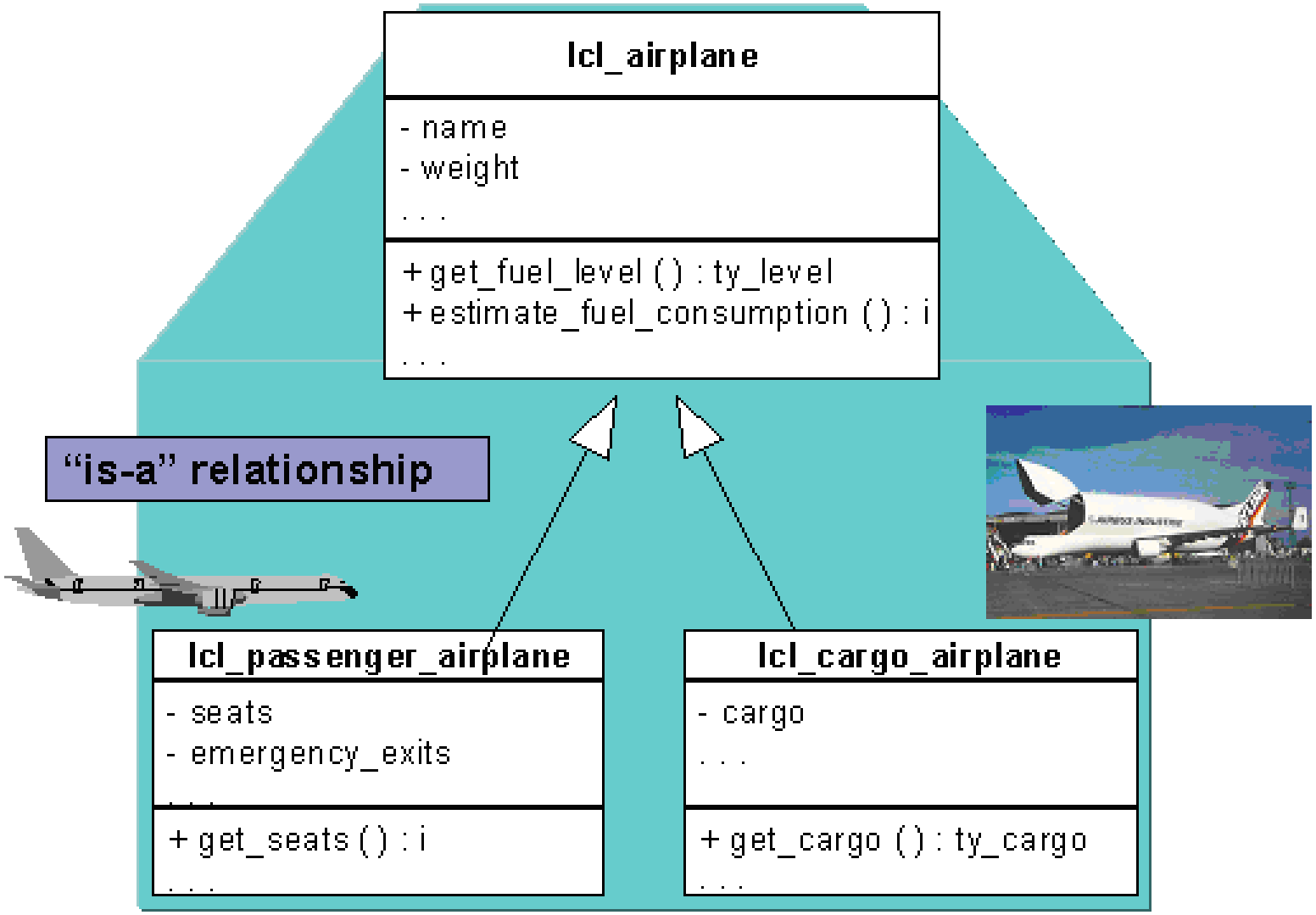
- The same namespace for
  - Attributes Names
  - Method Names
  - Event Names
  - Type Names
  - Constant Names
  - Alias Names

- There is a local namespace within methods

Generalisation / Specialisation

- Inheritance
- Cast
- Polymorphism
- Further characteristics of inheritance
- Interfaces
- Compound interfaces

Inheritance



Inheritance

- Inheritance is a relationship in which one class (the subclass) inherits all the main characteristics of another class (the superclass). The subclass can also add new components (attributes, methods, and so on) and replace inherited methods with its own implementations.
- Inheritance is an implementation relationship that emphasizes similarities between classes. In the example above, the similarities between the passenger plane and cargo plane classes are extracted to the airplane superclass. This means that common components are only defined/implemented in the superclass and are automatically present in the subclasses.
- The inheritance relationship is often described as an “is-a” relationship: a passenger plan is an airplane.

Inheritance

- Inheritance should be used to implement generalization and specialization relationships. A superclass is a generalization of its subclasses. The subclass in turn is a specialization of its superclasses.
- The situation in which a class, for example lcl_8, inherits from two classes (lcl_6 and lcl_7) simultaneously, is known as multiple inheritance. This is not realized in ABAP objects. ABAP objects only has single inheritance.
- However, you can simulate multiple inheritance in ABAP objects using interfaces.
- Single inheritance does not mean that the inheritance tree only has one level. On the contrary, the direct superclass of one class can in turn be the subclass of a further superclass. In other words: the inheritance tree can have any number of levels, so that a class can inherit from several superclasses, as long as it only has one direct superclass.
- Inheritance is a “one-sided relationship”: subclasses know their direct superclasses, but (super)classes do not know their subclasses.

Inheritance: Example

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    METHODS: get_fuel_level RETURNING VALUE(re_level) TYPE ty_level.

  PRIVATE SECTION.
    DATA: name   TYPE string,
          weight TYPE I.
ENDCLASS.
```

```
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.

  PUBLIC SECTION.
    METHODS: get_cargo RETURNING VALUE(re_cargo) TYPE ty_cargo.

  PRIVATE SECTION.
    DATA: cargo TYPE ty_cargo.

ENDCLASS.
```

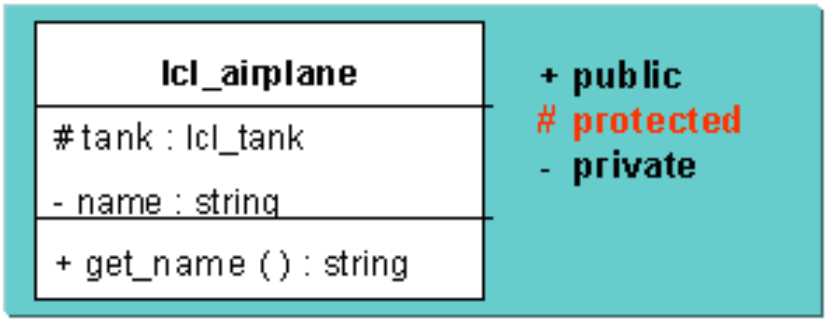
Relationships Between Superclasses and Subclasses

- Common components are only present once in the superclass
  - New components in the superclass are automatically available to the subclasses
  - Amount of new coding is reduced ("programming by difference")
- Subclasses are extremely dependent on superclasses
  - "White Box Re-use":
    Subclass must possess detailed knowledge of the implementation of the superclass

Inheritance and Visibility

- Public components
  - Visible to all
  - Direct access
- **Protected components**
  - **Only visible within their class and within the subclass**
- Private components
  - Only visible within the class
  - No access from outside the class,**not even from the subclass**

```
CLASS lcl_airplane DEFINITION.

  PUBLIC SECTION.
    METHODS get_name RETURNING
      VALUE(re_name) TYPE string.

  PROTECTED SECTION.
    DATA tank TYPE REF TO lcl_tank.

  PRIVATE SECTION.
    DATA name TYPE string.

ENDCLASS.
```



Inheritance and the (Instance) Constructor

```
CLASS lcl_airplane DEFINITION.
 PUBLIC SECTION.
  METHODS: CONSTRUCTOR IMPORTING
               im_name TYPE string.
ENDCLASS.
```

```
CLASS lcl_airplane IMPLEMENTATION.
  METHOD CONSTRUCTOR.
    name = im_name.
  ENDMETHOD.
ENDCLASS.
```

```
CLASS lcl_cargo_airplane DEFINITION INHERITING FROM lcl_airplane.
  PUBLIC SECTION.
    METHODS: CONSTRUCTOR IMPORTING im_name  TYPE string
                                   im_cargo TYPE ty_cargo.
  PRIVATE SECTION.
    DATA: cargo TYPE ty_cargo.
ENDCLASS.
```

```
CLASS lcl_cargo_airplane IMPLEMENTATION.
  METHOD CONSTRUCTOR.
    CALL METHOD SUPER->CONSTRUCTOR EXPORTING im_name = im_name.
    cargo = im_cargo.
  ENDMETHOD.
ENDCLASS.
```

Parameters and Create Object

- You must also consider the model described for instance constructors when using CREATE OBJECT. In this model, the constructor of the immediate superclass must be called before the non-inherited instance attributes can be initialized.
- There are basically two methods of creating an instance in a class using CREATE OBJECT:
  - 1. The instance constructor for that class has been defined (and implemented).
  - In this case, when you are using CREATE OBJECT, the parameters have to be filled according to the constructor interface, that is, optional parameters may, and non-optional parameters must be filled with actual parameters. If the constructor does not have any (formal) parameters, no parameters may or can be filled.
  - 2. The instance constructor for that class has not been defined.
  - In this case, you must search the inheritance hierarchy for the next highest superclass in which the instance constructor has been defined and implemented. Then, when you are using CREATE OBJECT, the parameters of that class must be filled (similarly to the first method above).

    If there is no superclass with a defined instance constructor, then no parameters may or can be filled.

Redefining Methods

- In ABAP Objects, you can not only add new components, but also provide inherited methods with new implementations. This is known as redefinition. You can only redefine (public and protected) instance methods, other components (static methods, attributes and so on) cannot be redefined. Furthermore, implementation is restricted to (re-)implementation of an inherited method; you cannot change method parameters (signature change).
- You also cannot redefine a class's (instance) constructor.
- In UML, the redefinition of a method is represented by listing the method again in the subclass. Methods (and all other components) that are inherited but not redefined are not listed in the subclass, as their presence there is clear from the specialization relationship.
- You should not confuse redefinition with "overlaying". This describes the ability of a class to have methods with the same name but a different signature (number and type of parameters). This option is not available in ABAP Objects.

Parameters and Create Object

- If no instance constructor has been defined for a class, then a default constructor, which is implicitly always present is used. This default constructor calls the constructor from the immediate superclass.

Compatibility and Narrowing Cast

- One of the significant principles of inheritance is that an instance from a subclass can be used in every context in which an instance from the superclass appears. This is possible because the subclass has inherited all components from the superclass and therefore has the same interface as the superclass. The user can therefore address the subclass instance in the same way as the superclass instance.
- Variables of the type “reference to superclass” can also refer to subclass instances at runtime.

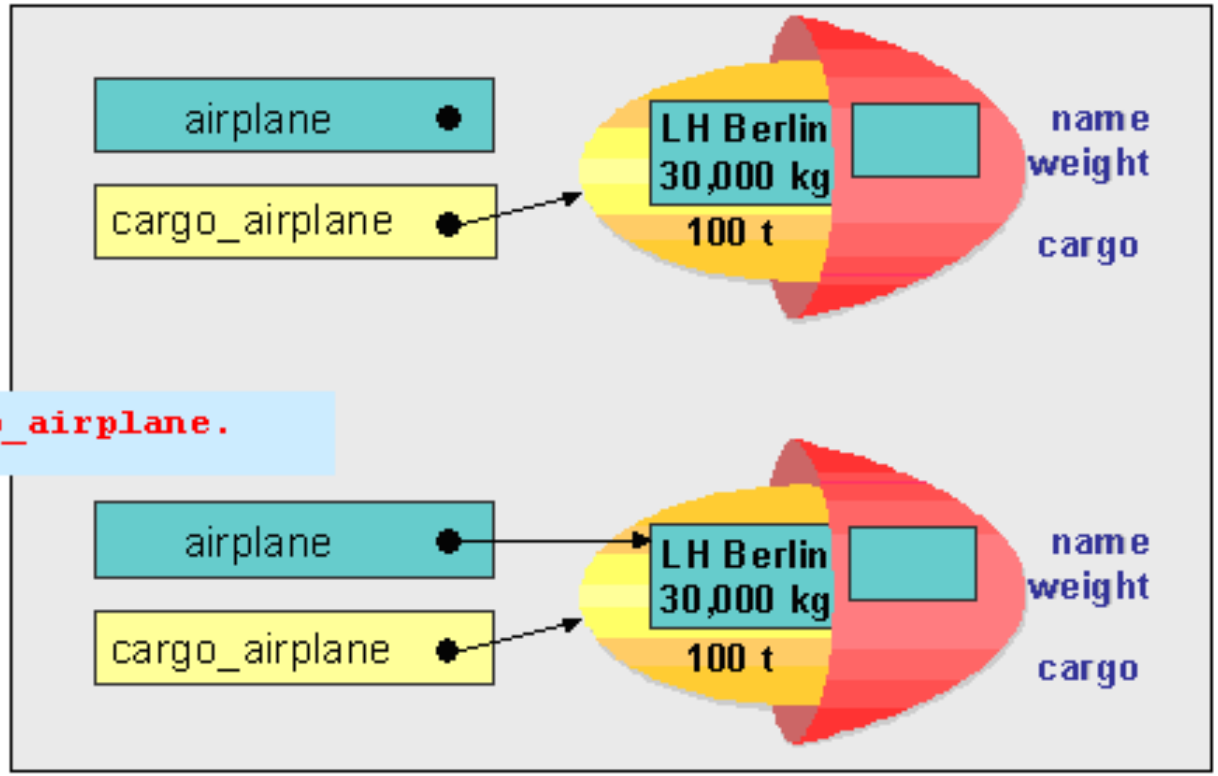
Compatibility and Narrowing Cast

- The assignment of a subclass instance to a reference variable of the type “reference to superclass” is described as a narrowing cast, because you are switching from a view with more detail to a view with less detail.
- The description “up-cast” is also used.
- What is a narrowing cast used for? A user who is not interested in the finer points of cargo or passenger planes (but only, for example, in the tank gauge) does not need to know about them. This user only needs to work with (references to) the lcl_airplane class. However, in order to allow the user to work with cargo or passenger planes, you would normally need a narrowing cast.

Principals of Narrowing Cast

```
DATA: airplane       TYPE REF TO lcl_airplane,
      cargo_airplane TYPE REF TO lcl_cargo_airplane.
```

```
CREATE OBJECT cargo_airplane.
```




```
airplane = cargo_airplane.
```

Static and Dynamic Types

- A reference variable always has two types: static and dynamic:
  - The static type of a reference variable is determined by variable definition using TYPE REF TO. It cannot and does not change. It establishes which attributes and methods can be addressed
  - The dynamic type of a reference variable is the type of the instance currently being referred to, it is therefore determined by assignment and can change during the program run. It establishes what coding is to be executed for redefined methods.
- The reference ME can be used to determine the dynamic type in the debugger.

Polymorphism and Inheritance

- When objects from different classes react differently to the same method call, this is known as polymorphism. To do this, the classes implement the same method in different ways. This can done using inheritance, by redefining a method from the superclass in subclasses and implementing it differently. Interfaces are also introduced below: they too can enable polymorphic behavior!
- When an instance receives a message to execute a particular method, then that method is executed if it has been implemented by the class the instance belongs to. If the class has not implemented that method, but only inherited and not redefined it, then a search up through the inheritance hierarchy is carried out until an implementation of that method is found.

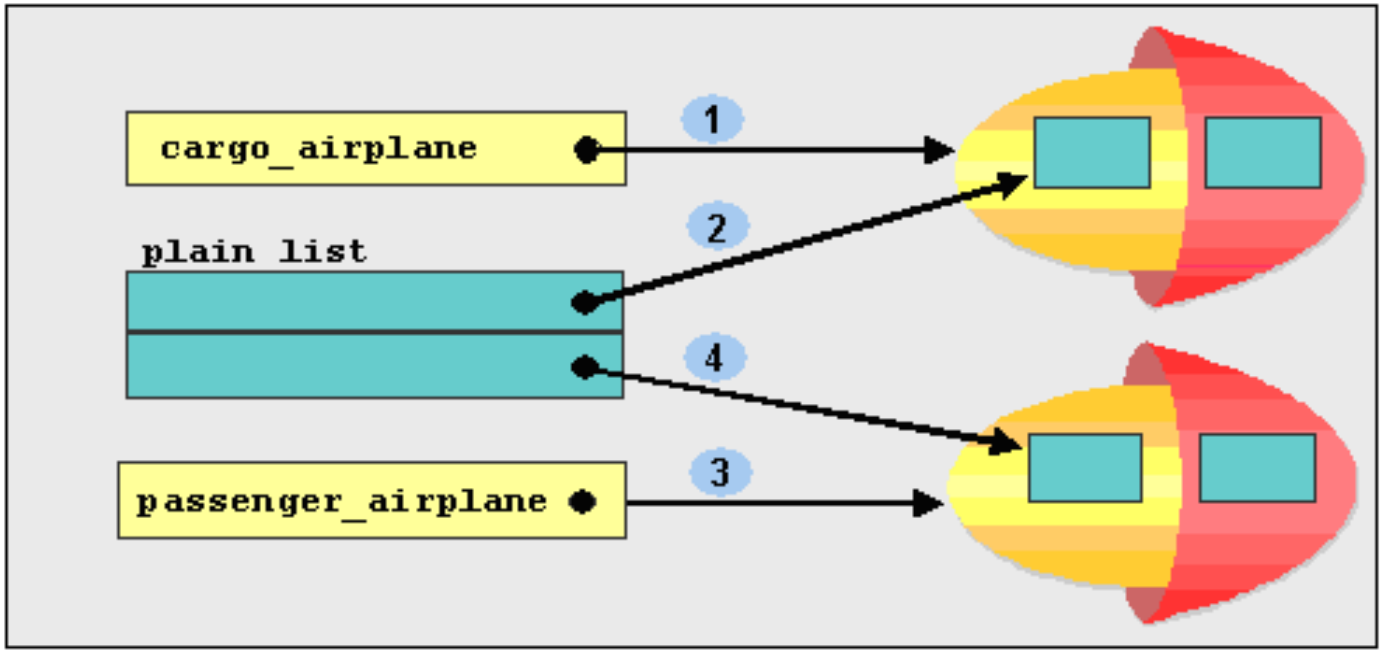
Polymorphism and Inheritance

- Technically speaking, the dynamic type of the reference variable, not the static type, is used to search for the implementation of a method. In the above example of CALL METHOD plane->estimate_fuel_consumption, the class of the instance that plane actually refers to is used to search for the implementation of estimate_fuel_consumption; The static type of plane, which is always ‘REF TO lcl_airplane’ is not used.
- Polymorphism is one of the main strengths of inheritance: the user can work in the same way with different classes, regardless of their implementation. The search for the right implementation of a method is carried out by the runtime system, not the user!

Polymorphism: Example

```
DATA: cargo_plane     TYPE REF TO lcl_cargo_airplane,
      passenger_plane TYPE REF TO lcl_passenger_airplane,
      plane_list      TYPE TABLE OF REF TO lcl_airplane.
```

```
1 CREATE OBJECT: cargo_plane.
2 APPEND cargo_plane TO plane_list.
3 CREATE OBJECT passenger_plane.
4 APPEND passenger_airplane TO plane_list.
```



Abstract Classes and Methods

- You cannot instantiate objects in an abstract class. This does not, however, mean that references to such classes are meaningless. On the contrary, they are very useful, since they can (and must) refer to instances in subclasses of the abstract class during runtime. The CREATE-OBJECT statement is extended in this context. You can enter the class of the instance to be created explicitly:
- CREATE OBJECT <RefToAbstractClass> TYPE <NonAbstractSubclassName>.
- Abstract classes are normally used as an incomplete blueprint for concrete (that is, non-abstract) subclasses, in order to define a uniform interface, for example.

Abstract Classes and Methods

- Abstract instance methods are used to specify particular interfaces for subclasses, without having to immediately provide implementation for them. Abstract methods need to be redefined and thereby implemented in the subclass (here you also need to include the corresponding redefinition statement in the DEFINITION part of the subclass).
- Classes with at least one abstract method are themselves abstract
- Static methods and constructors cannot be abstract (they cannot be redefined).

Final Classes and Methods

- A final class cannot have subclasses, and can protect itself in this way against (uncontrolled) specialization.
- A final method in a class cannot be redefined in a subclass, and can protect itself in this way against (uncontrolled) redefinition.
- Some characteristics:
  - A final class implicitly only contains final methods. You cannot enter FINAL explicitly for these methods in this case.
  - Methods cannot be both final and abstract.
  - Classes, on the other hand, can usefully be both abstract and final: only static components can be used there.

Using Inheritance

- Classes can be extended using specialization
- Re-use
- Polymorphic behavior through redefinition
  - No need to program CASE structures

- Inheritance is often used incorrectly.
  - To simply recycle coding
  - Instead of additional attributes/aggregation/role concepts
  - The use of inheritance does not always correspond to expectations in the real world

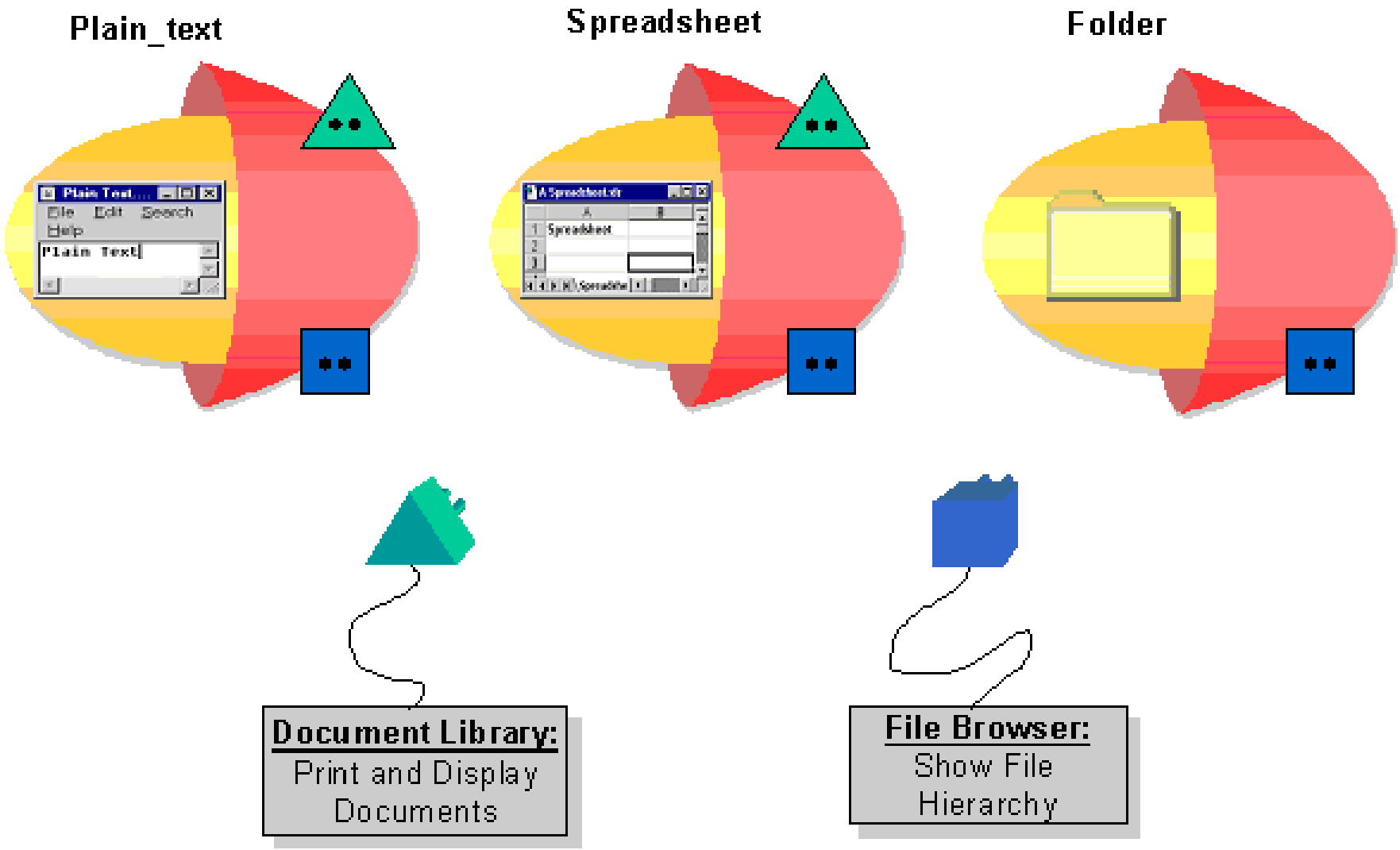
Using Inheritance

- Using inheritance instead of attributes, or a misunderstanding of inheritance as an “is-one” relationship often leads to the following kind of design: the superclass “car” has the subclasses “red car”, “green car”, and so on. These subclasses all have an identical structure and identical behavior.
- As an instance cannot change its class, in circumstances like the following, you should not use inheritance directly, but use additional attributes to differentiate between cases the class “employee” has the subclasses “full-time employee” and “part-time employee”. What happens when a part-time employee becomes a full-time employee? A new full-time-employee object would have to instantiated and all the information copied from the part-time-employee object. However, users who still have a reference to the part-time-employee instance would then be working with a part-time-employee object that logically does not exist anymore!
- The use of inheritance does not always correspond to expectations in the real world: for example, if the class ‘square’ inherits from the class ‘rectangle’, the latter will have two separate methods for changing length and width, although the sides of the square actually need to be changed by the same measurement.

Interfaces

- In ABAP objects, interfaces are implemented in addition to and independently of classes. Interfaces exclusively describe the external point of contact of a class, but they do not contain their own implementation part.
- Interfaces are usually defined by a user. The user describes in the interface which services (technical and semantic) it needs in order to carry out a task. The user never actually knows the providers, but communicates with them through the interface. In this way the user is protected from actual implementations and can work in the same way with different classes/objects, as long as they provide the services required (this is polymorphism using interfaces).

Interfaces


Plain_text
Spreadsheet
Folder
Plain Text
File Edit Search
Help
Plain Text
Spreadsheet
Document Library:
Print and Display Documents
File Browser:
Show File Hierarchy

Defining and Implementing an Interface

- Interface only has a declaration
- An interface corresponds to an abstract class that only contains abstract methods
- Interfaces are implemented in classes
- Interfaces do not have visibility sections

```
INTERFACE lif_document.
  DATA:    author TYPE REF TO lcl_author.
  METHODS: print,
           display.
ENDINTERFACE.
```

```
CLASS lcl_text_document DEFINITION.
  PUBLIC SECTION.
    INTERFACES lif_document.
    METHODS: display.
ENDCLASS.
```

```
CLASS lcl_text_document IMPLEMENTATION.
  METHOD lif_document~print.
  ENDMETHOD.
  METHOD lif_document~display.
  ENDMETHOD.
  METHOD display.
  ENDMETHOD.
ENDCLASS.
```

Defining and Implementing an Interface

- In ABAP objects, the same components (attributes, methods, constants, types, alias names) can be defined in an interface in largely the same way as in classes. However, interfaces do not have component visibility sections.
- Interfaces are implemented in classes.
  - The interface name is listed in the definition part of the class. Interfaces can only be implemented ‘publicly’ and are therefore always in the PUBLIC SECTION (this is only valid as of release 4.6). If you do not do this, you risk multiple implementations, if a superclass and a subclass both implement the same interface privately.
  - The operations defined in the interface are implemented as methods of a class. A check is carried out to ensure that all the methods defined in the interfaces are actually present in the implementation part of the class (for global interfaces, a missing or superfluous implementation of an interface method results in a ToDo warning).
  - The attributes, events, constants and types defined in the interface are automatically available to the class carrying out the implementation.

Defining and Implementing an Interface

- Interface components are addressed in the class carrying out the implementation by prefixing the interface name, followed by a tilde (the interface resolution operator): <interfacename>~<componentname>.

Working With Interface Components

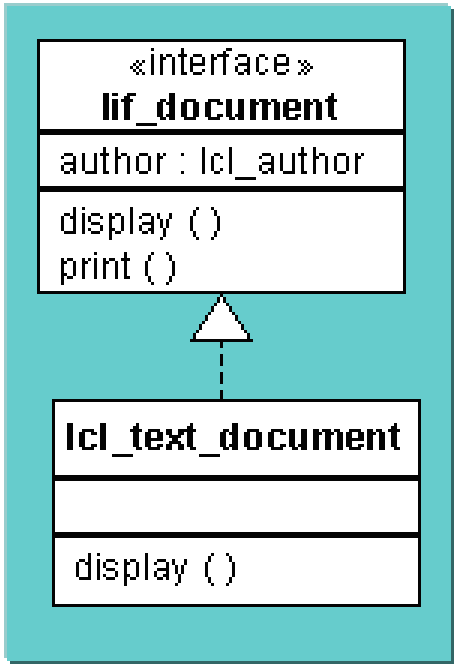
- The interface resolution operator enables you to access interface components using an object reference belonging to the class implementing the interface in exactly the same way as the method definition in the implementation part of the class.
- This allows you to differentiate between components defined in the interface and components of the same name that are defined in the class itself. This is caused by the shared namespace.

Working With Interface Components

```
CLASS lcl_text_document IMPLEMENTATION.
  METHOD lif_document~print. ...
  ENDMETHOD.
  METHOD lif_document~display.    ...
  ENDMETHOD.
  METHOD display. ...
  ENDMETHOD.
ENDCLASS.
```

```
DATA: text_doc TYPE REF TO lcl_text_document.
```

```
CREATE OBJECT text_doc.

CALL METHOD text_doc->lif_document~print.
CALL METHOD text_doc->lif_document~display.
CALL METHOD text_doc->display.
```



Polymorphism and Interfaces

- Polymorphism can also be used for interfaces: you can use interface references to call methods that can have a different implementation depending on the object behind the reference.
- The dynamic type, not the static type of the reference variable is used to search for the implementation of a method. CALL METHOD document->display above therefore uses the class of the instance that document actually refers to to search for the implementation of display. The static type for document, which is always ‘REF TO lif_doc’ is not used.

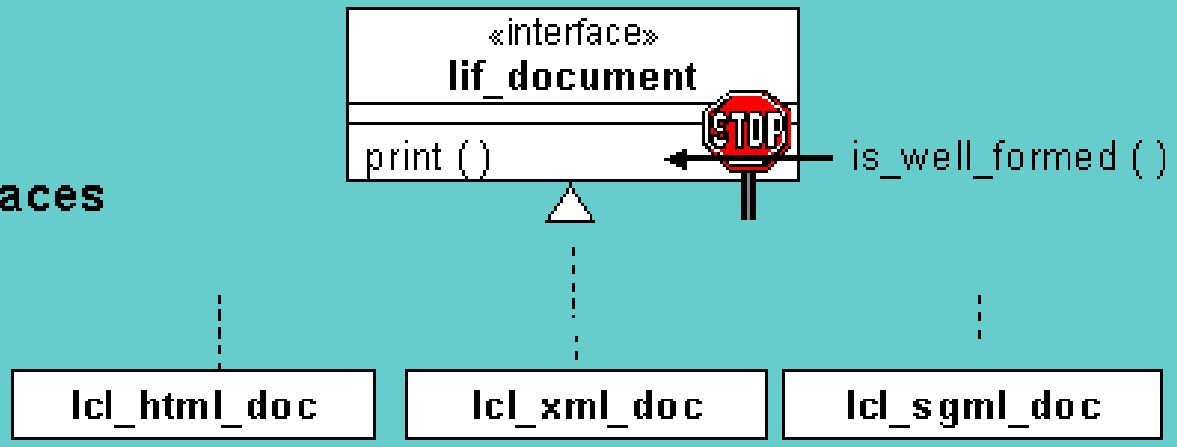
Differences Between Polymorphism and Inheritance & Polymorphism and Interfaces

- Polymorphism and Inheritance
  - can only be used with objects from classes that are connected by an inheritance hierarchy

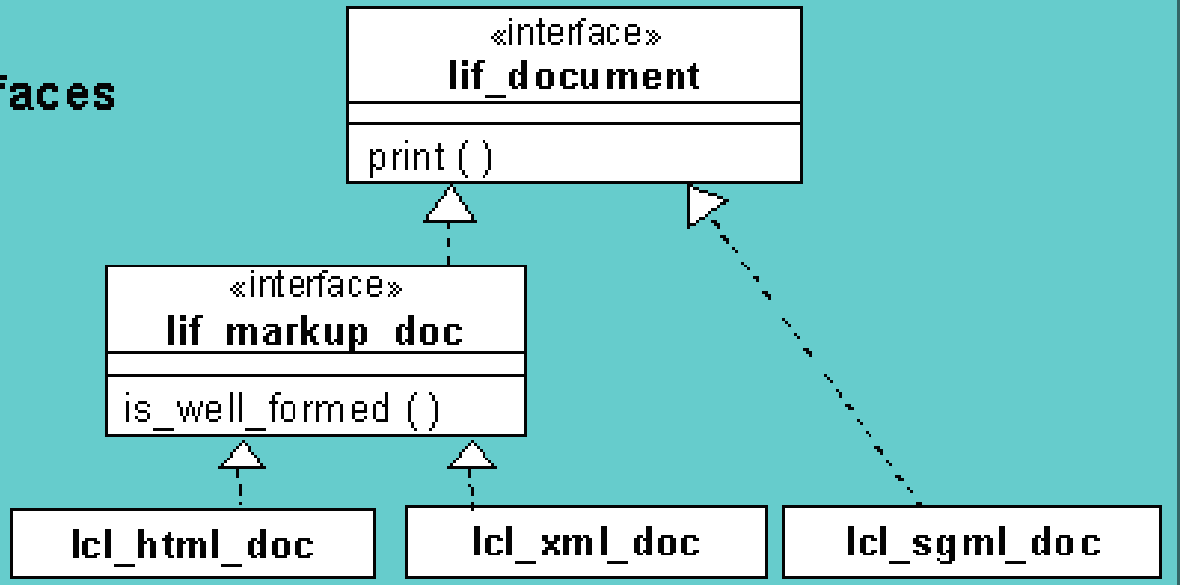
- Polymorphism and interfaces
  - Can be used with objects from any class, as long as these classes have implemented the corresponding interface

Compound Interfaces

- **Problem:**
  extending interfaces



- **Solution:**
  compound interfaces



Compound Interfaces

- Changes to an interface usually invalidate all the classes implementing it.
- ABAP objects contains a composition model for interfaces. A compound interface contains other interfaces as components (component interfaces) and is therefore a summarized extension of these component interfaces. An elementary interface does not itself contain other interfaces.
- One interface can be used as a component interface in several compound interfaces.
- Compound interfaces in ABAP objects can always be seen as specializations of their component interfaces.

Compound Interfaces: Example

```
INTERFACE lif_doc.
  METHODS edit.
ENDINTERFACE.
```

```
INTERFACE lif_markup_doc.
  INTERFACES lif_doc.
  METHODS is_well_formed.
ENDINTERFACE.
```

```
CLASS lcl_html_doc DEFINITION.
  PUBLIC SECTION.
    INTERFACES lif_markup_doc.
ENDCLASS.
CLASS lcl_html_doc IMPLEMENTATION.
  METHOD lif_doc~edit.
  ENDMETHOD.
  METHOD lif_markup_doc~is_well_formed.
  ENDMETHOD.
ENDCLASS.
```

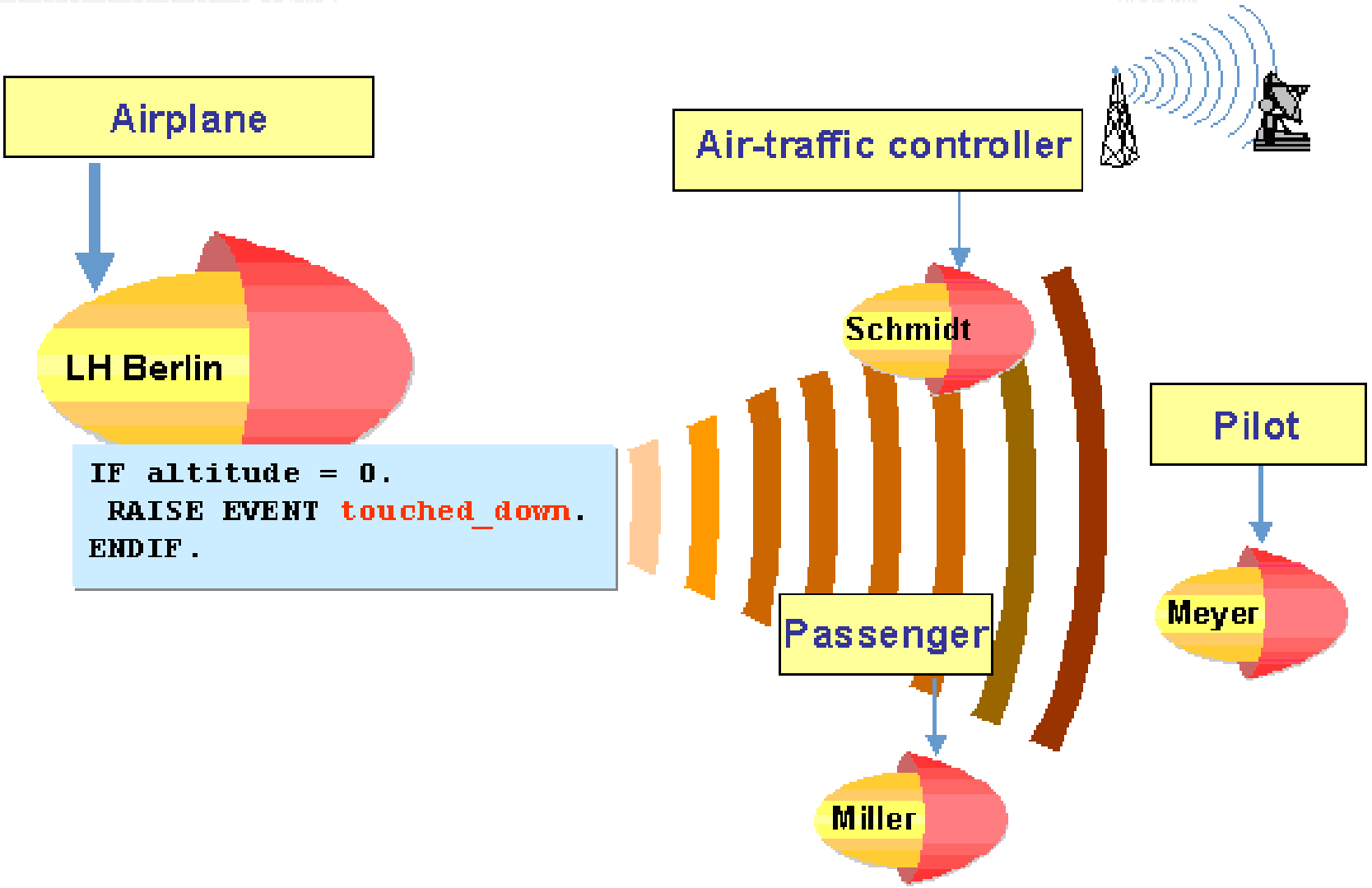
```
DATA: i_doc        TYPE REF TO lif_doc,
      i_markup_doc TYPE REF TO lif_markup_doc,
      html_doc     TYPE REF TO lcl_html_doc.
```

```
i_doc = i_markup_doc.          "Narrowing Cast

CALL METHOD i_markup_doc->lif_doc~edit.
*CALL METHOD i_doc->edit.
*CALL METHOD html_doc->lif_doc~edit.

i_markup_doc ?= i_doc.         "Widening Cast
```

Using Interfaces

- Interfaces are the means of choice for describing external points of contact, without linking them to a type of implementation. An extra layer is introduced between the client and the server to protect the client explicitly from the server, thereby making it much more independent!
- Interfaces enable you to work uniformly with different classes (providers). In particular, they always ensure polymorphic behaviour as they do not have their own implementation, but instead allow the providers to carry it out.
- The definition of an interface is always an abstraction: the user wants to handle various providers in the same way and must therefore abstract concrete implementations to a description of the services required to fulfil the task.
- You can also use interfaces to achieve multiple inheritance by defining the functionality to be inherited by a second class as an interface that the inheriting class then has to implement.

Events


Airplane
LH Berlin
IF altitude = 0.
RAISE EVENT touched_down.
ENDIF.
Air-traffic controller
Schmidt
Pilot
Meyer
Passenger
Miller

Events

- By triggering an event, an object or a class announces a change of state, or that a certain state has been achieved.
- In the above example, the airplane class triggers the event ‘touched_down’. Other classes subscribe to this event and process it. The air-traffic controller marks the plane as landed on the list, the pilot breathes a sigh of relief and the passenger, Mr. Miller, applauds.
- Note:

  The events discussed here are not ABAP events such as INITIALIZATION,

  START-OF-SELECTION, and so on.

Events: Characteristics and Uses

- Events link objects or classes more loosely than direct method calls do. Method calls establish precisely when and in which statement sequence the method is called. However, with events, the reaction of the object to the event is determined by the triggering of the event itself.
- Events are most often used in GUI implementations.
- Other external object models, such as COM, ActiveX controls etc, also provide events.

Triggering and Handling Events

- Triggering Events
  - Class defines event
    (EVENTS, CLASS_EVENTS)
  - Object or class triggers event
    (RAISE EVENT)

- Handling events
  - Event handler class defines and implements event handler method
    ([CLASS-]METHODS...FOR EVENT ... OF ...)
  - "Event handler object" or handler class registers itself to specific object/class events at runtime
    (SET HANDLER)

Triggering and Handling Events

- At the moment of implementation, a class defines its
  - Instance events (using the statement EVENTS) and
  - Static events (using the statement CLASS-EVENTS)
- Classes or their instances that receive a message when an event is triggered at runtime and want to react to this event define event handler methods.

  Statement : (CLASS-)METHODS <handler_method> FOR EVENT <event> OF <classname>.
- These classes or their instances register themselves at runtime to one or more events.

  Statement : SET HANDLER <handler_method> FOR <reference>. (For instance events)

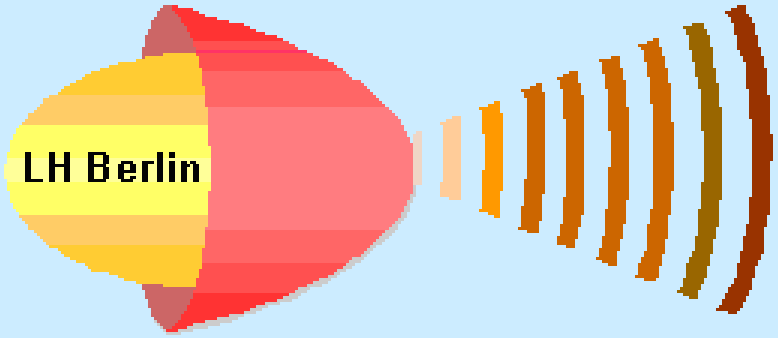
  SET HANDLER <handler_method>. (For static events).
- A class or an instance can trigger an event at runtime using the statement RAISE EVENT.

Defining and Triggering Events: Syntax

```
CLASS <classname> DEFINITION.
  EVENTS: <event> EXPORTING VALUE(<ex_par>) TYPE type.
```

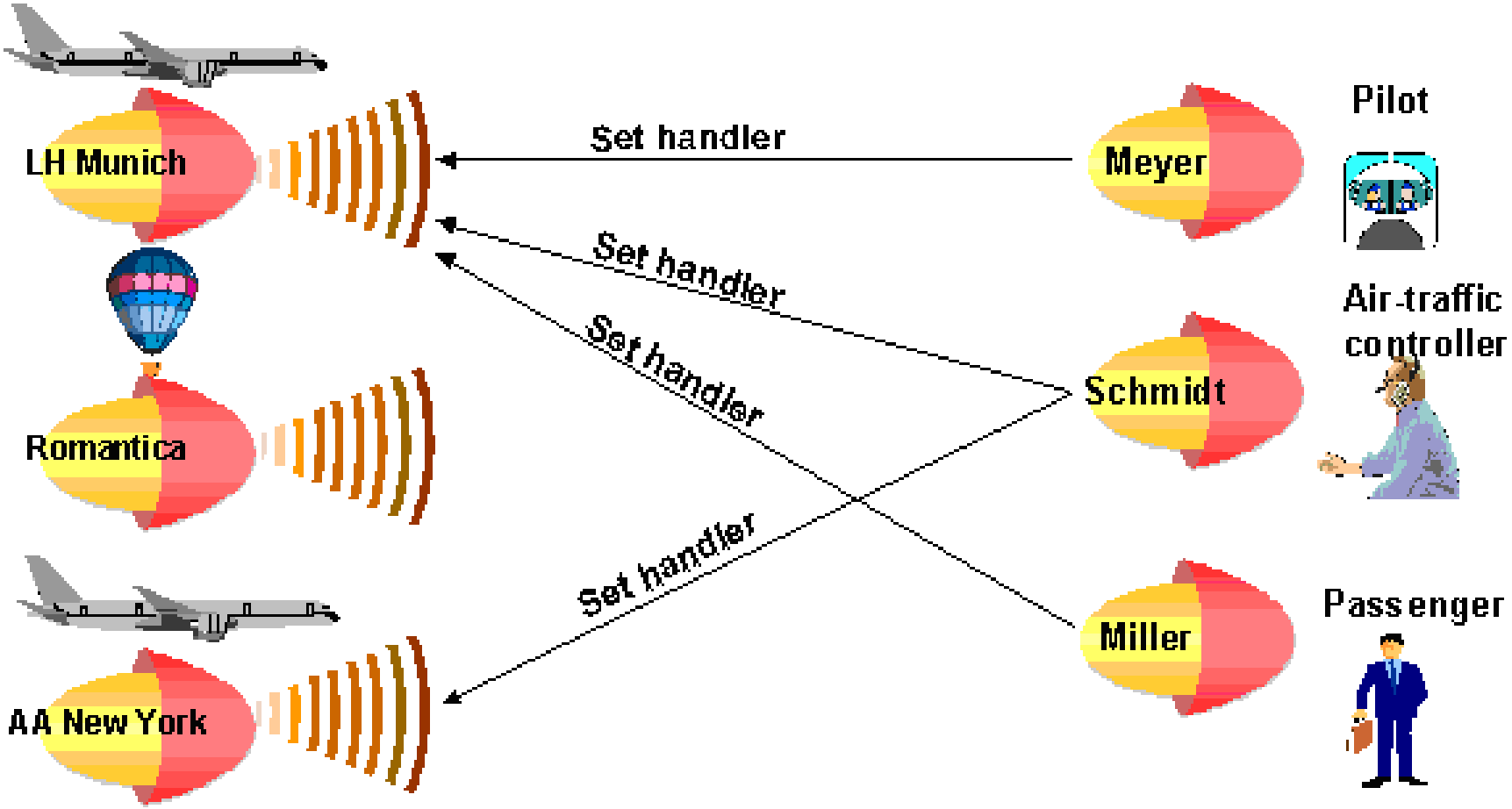
```
CLASS <classname> IMPLEMENTATION.
  METHOD  <m>.
   RAISE EVENT <event> EXPORTING <ex_par> = <act_par>.
```

```
CLASS lcl_airplane DEFINITION.
  PUBLIC SECTION.
    METHODS arrive_at_airport.
    EVENTS touched_down EXPORTING VALUE(ex_name) TYPE string.
  PRIVATE SECTION.
    DATA: name TYPE string.
ENDCLASS.

CLASS cl_airplane IMPLEMENTATION.
  METHOD arrive_at_airport.
    ...
    RAISE EVENT touched_down EXPORTING ex_name = name.
  ENDMETHOD.
ENDCLASS.
```



Handling and Registering Events

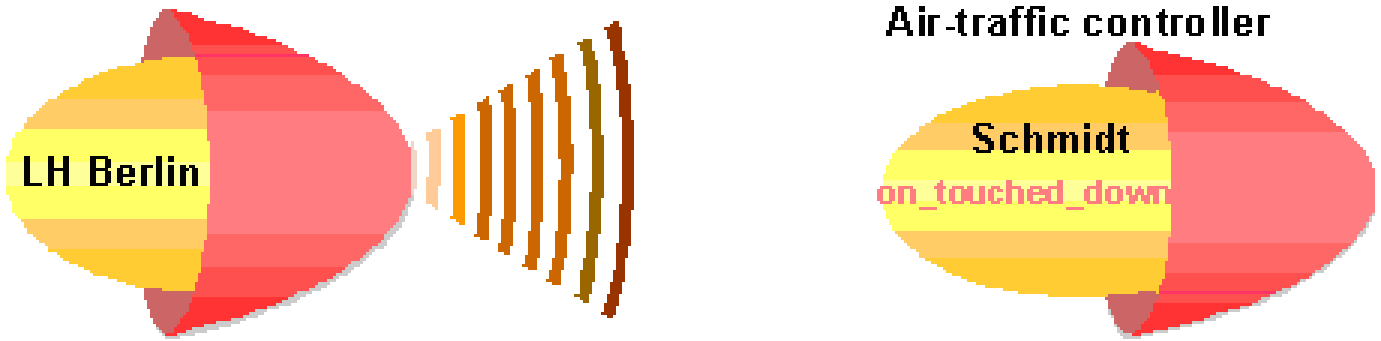
```
METHODS on_touched_down FOR EVENT touched_down OF cl_airplane.
```



Handling and Registering Events: Syntax

```
CLASS <class_handle> DEFINITION.
  METHODS: <on_event> FOR EVENT <event>
                      OF <classname> | <interface>
                      IMPORTING <ex_par1> ... <ex_parN> SENDER.
```

```
CLASS lcl_air_traffic_controller DEFINITION.
  ...
  PRIVATE SECTION.
    METHODS: on_touched_down FOR EVENT touched_down OF lcl_airplane
                             IMPORTING ex_name
                                       SENDER.
ENDCLASS.
```



Event Handling Characteristics

- Event handling is sequential.
- Sequence in which event handler methods are called is not defined.
- As far as the Garbage Collector is concerned, registration has the same effect as a reference to the object registered.
  - Registered objects are never deleted.

- Immediate effects of SET HANDLER on event handler methods:
  - Newly registered event handlers are also executed.
  - Deregistered handlers may already have been executed.

Event Handling Characteristics contd.

- If several objects have registered themselves for an event, then the sequence in which the event handler methods are called is not defined, that is, there is no guaranteed algorithm for the sequence in which the event handler methods are called.
- If a new event handler is registered in an event handler method for an event that has just been triggered, then this event handler is added to the end of the sequence and is then also executed when its turn comes.
- If an existing event handler is deregistered in an event handler method, then this handler is deleted from the event handler method sequence.

Defining and Triggering Events: Syntax

- Both instance and static events can be triggered in instance methods.
- Only static events can be triggered in static methods.
- Events can only have EXPORTING parameters which must be passed by value.
- Triggering an event using the statement RAISE EVENT has the following effect:
  - The program flow is interrupted at that point
  - The event handler methods registered to this event are called and processed
  - Once all event handler methods have been executed, the program flow starts again.
- If an event handler method in turn triggers an event, then the program flow is again interrupted and all event handler methods are executed (nesting).

Events and Visibility

- Events are also subject to the visibility concept and can therefore be either public, protected or private. Visibility establishes authorization for event handling :
  - all users
  - only users within that class or its subclasses
  - only users in that class.
- Event handler methods also have visibility characteristics. Event handler methods, however, can only have the same visibility or more restricted visibility than the events they refer to.
- The visibility of event handler methods establishes authorization for SET-HANDLER statements: SET HANDLER statements can be made
  - anywhere
  - in that class and its subclasses
  - only in that class

Local Classes /Interfaces

- Local classes/interfaces are only known within the program in which they are defined and implemented.
- Local classes/interfaces are not stored in the repository (no TADIR entry). There is no “global” access to these classes/interfaces (for example, from other programs).
- If a local class is implemented in an include which is then embedded in two different programs, then references to the “same” class still cannot be exchanged at runtime. Two classes that do not conform to type are created at runtime.

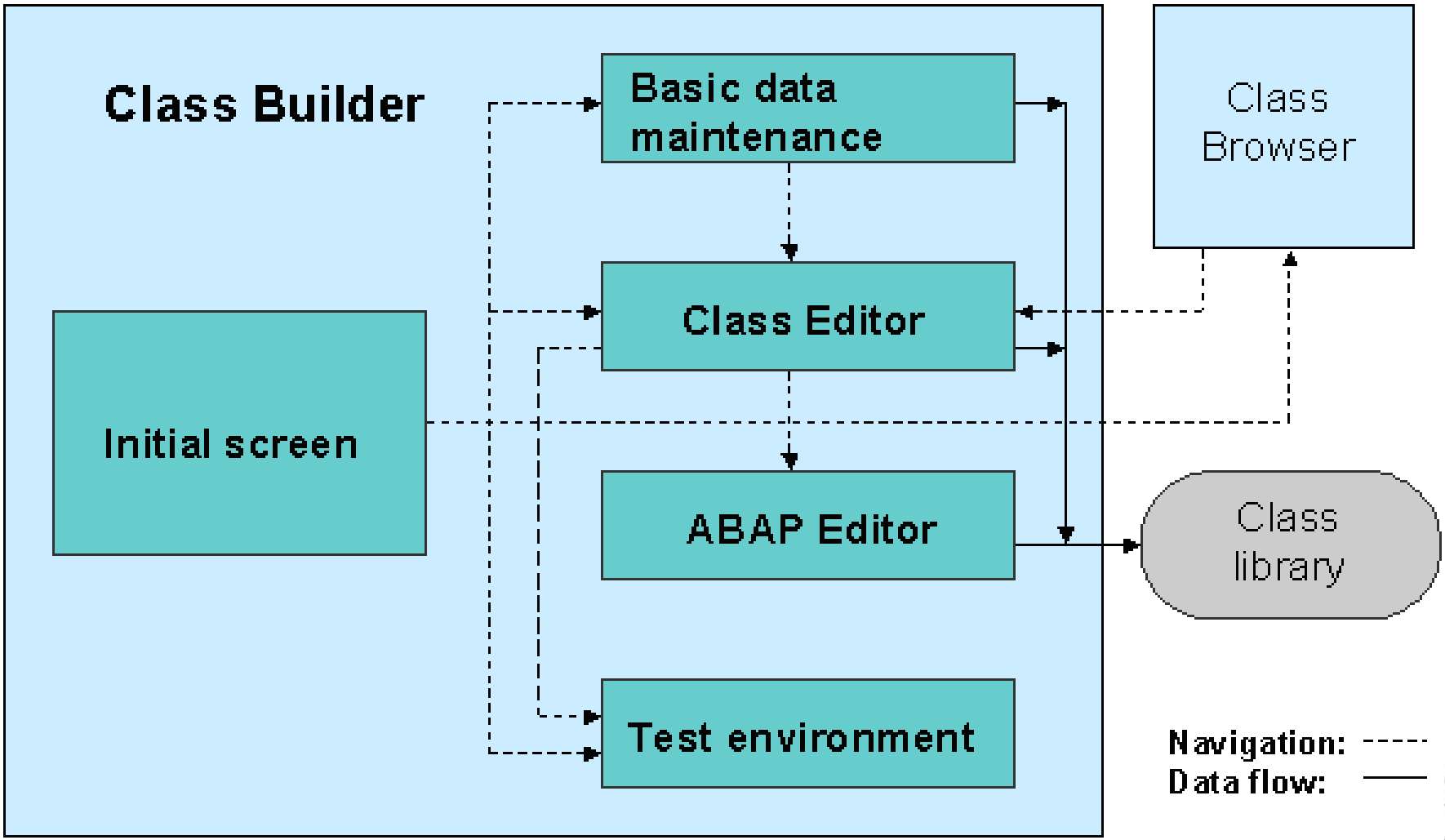
Global Classes / Interfaces

- Unlike local in program classes/interfaces, global classes/interfaces can be created and implemented using the ABAP workbench tool class builder or transaction SE24. These classes/interfaces are then available to all developers.
- Global class and interface names share the same namespace.
- Global classes/interfaces have a TADIR entry: R3TR CLASS <name>
- The smallest transport unit is method LIMU METH.

Class Builder

- The class builder is a tool in the ABAP workbench that is used to create, define and test global ABAP classes and interfaces.

Class Builder: Structure



Class Builder: Structure

- In the initial screen, select the object type you want to work with - class or interface. Then choose one of the display, change, create or test functions.
- In the initial screen you have the choice of viewing the contents of the R/3 class library using the class browser or going straight to basic data maintenance of the object types and the class editor, where you can define the object types and their components. The object type definition can be immediately followed by method implementation in the ABAP editor. You can also access the test environment from the initial screen or from the class editor.

TATA CONSULTANCY SERVICES

TATA

Thank You!